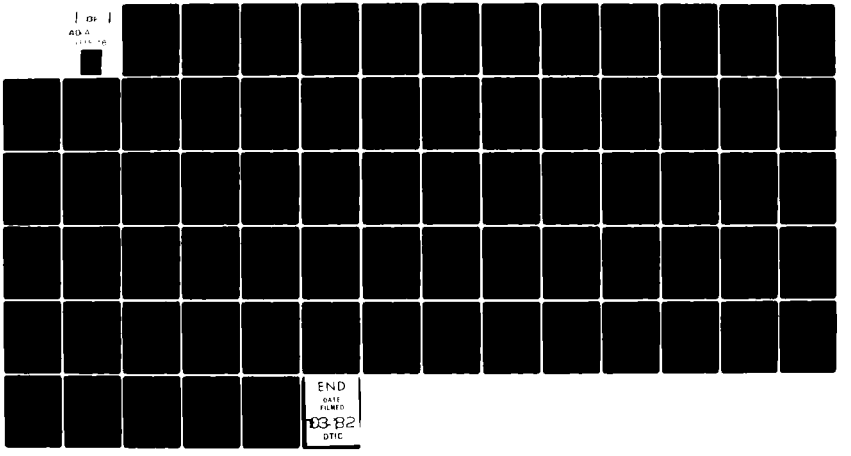
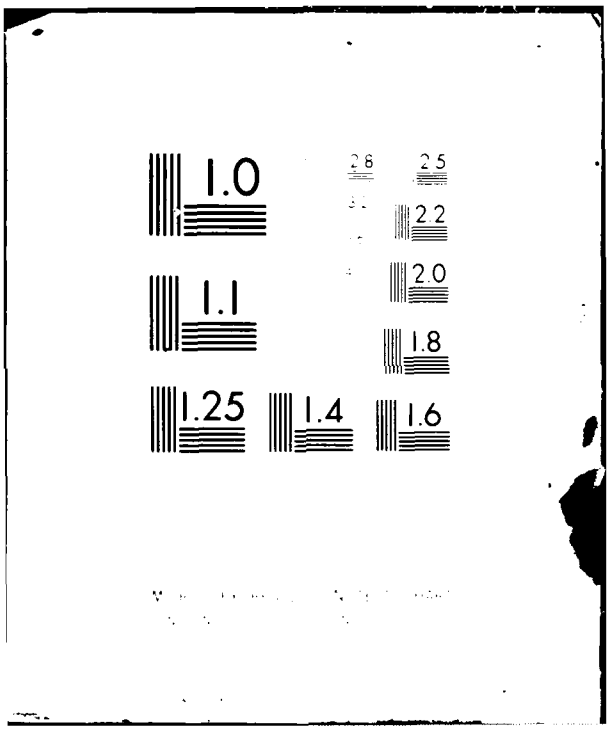
AD-A111 576 FORD AEROSPACE AND COMMUNICATIONS CORP PALO ALTO CA W--ETC F/G 9/2
KSOS STANDARDS AND PROCEDURES FOR PROGRAMS AND FORMAL SPECIFICA--ETC(U)
DEC 80 MDA903-77-C-0333
UNCLASSIFIED WDL-TR8999 NL




1.0
2.8
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

WDL-TR8999
December 1980

(2)

SECURE MINICOMPUTER OPERATING SYSTEM (KSOS)

STANDARDS AND PROCEDURES FOR PROGRAMS AND FORMAL SPECIFICATIONS

Department of Defense Kernelized Secure Operating System

Contract MDA 903-77-C-0333
CDRL 0002AP

Prepared for:

Defense Supply Service - Washington
Room 1D245, The Pentagon
Washington, DC 20310

Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

3939 Fabian Way
Palo Alto, California 94303

KSOS Standards and Procedures

Ford Aerospace and Communications Corporation

A number of guidelines have been established for KSOS development. They have provided a philosophical and conceptual framework for the KSOS program. By furnishing general guidelines, rather than a morass of rules to be slavishly obeyed, the pitfalls of over-structuring and repression of creative thought were avoided. Thus a workable level of uniformity of the development environment was achieved with tolerable overhead. Such uniformity contributes significantly to efficiency of the development process, and to reliability and maintainability of the product.

Standards for KSOS are categorized as follows:

- operational standards
- language style
- documentation
- testing,

Specific KSOS standards are given here as appendices containing the various documents defining the standards. The appendices follow the categories previously described:

OPERATIONAL

- File Naming Conventions
- KSOS Development Directory Structure

LANGUAGE

- SPECIAL Language Specification Standards for KSOS
- HDL Standards for KSOS
- Modula Language Programming Standards for KSOS
- UNIX Coding Guidelines
- C Language Programming Standards for KSOS
- Definition of Boolean Type in KSOS UNIX*tm Emulator Code
- Standard Headers

DOCUMENTATION

- Instructions for writing KSOS Kernel Manual Pages

TESTING

- Standard Test Frame Skeleton

OPERATIONAL

Ford Aerospace

subject: File Naming Conventions

date: January 12, 1981

from: Karl C. Kelley

TM: File Naming Conventions

PROGRAMMER'S NOTES

The intention here is to record a set of conventions for the orderly naming of files used in the KSOS project. This verison of the conventions was preceeded by a draft version and invitation for comments from those affected. All of those comments were heard, and some of them resulted in modifications. The result is workable for the whole group, but not optimal for anyone.

1. First Principles

The Unix system we have inherited is itself inconsistant a good deal of the time. Any conventions we decide upon must be constrained to work within the current Unix framework. For example, we are not free to start naming C program source files in any way that does not end in the preordained ".c".

Lets examine briefly the rationale for that convention and see if it leads us to any conclusion about our own conventions. Certain files which are prepared exclusively by the system have constructed in the front of them a special code or "magic number" which indicates the nature of the file (such as compiler outputs, archives, etc.) However, since the source for the compiler is simply a text file, and there are many text files prepared by many different tools, they (someone, at some past time) decided to use the suffix ".c" as an indicator of C-language source. The C-compiler, and a small number of other utilities, actually look at the name suffix to assure itself that "this file is intended for processing by this utility." This suggests that the original intent of using a file name suffix was to make it clear how the file is expected to be used. Notice that it in no way attempts to tell what the C program is about. That function is left to the earlier portions of the name.

Taken by itself, this argument would lead to:

THE FIRST PRINCIPLE OF NAMING FILES

> A period (".") in a file name introduces a suffix which is indicative of the intended normal use of the file.

There are a number of corollaries to this principle:

1. The file name suffix should be as short as possible to convey its meaning unambiguously.

2. The suffixes used should be agreed to by all the tools and users of tools to which it applies.

3. The period should not be used in a directory name.

4. The period should never be used to introduce a suffix implying anything about the topical content of the file.

The observant reader will immediately notice that the creators of SCCS were not cognizant of these principles when they introduced the notion of a prefix terminated by this same character to mark files under control of SCCS. This introduces the following:

THE LONE EXCEPTION TO
THE FIRST PRINCIPLE OF NAMING FILES

> SCCS file names are the only names in which a period is used for a purpose other than to indicate how the file is intended to be used. In general, the "s." prefix marks it as an SCCS file and any other period introduces a filename suffix in accordance with the "normal" practice.

It follows from this that other SCCS produced files will also use a prefix to identify themselves. There is after all the "p." convention used by SCCS. Obviously we will let these stand as acceptable and just adjust to their presence.

Life would be simpler for some of the tools we use and contemplate building, if a corollary to this exception were also followed:

COROLLARY TO LONE EXCEPTION

> SCCS files will always be named with an appropriate "period-introduced" suffix to indicate how the file is intended to be used.

There are rare instances when it can be considered acceptable to use multiple suffixes to indicate that a file is meant to be processed by one tool and its result is meant to be processed by another tool. For example the file alpha.m.m4 would indicate that m4 is to be run on the file and it will produce a file for input to mm. The binding principle is that the period-introduced suffix indicates intended use. The suggestion is that for simply separating abbreviations, the underscore character is more suitable.

1.1 Examples

The following illustrate accepted practice:

bio.c	indicates source for a C program
s.bio.c	indicates SCCS version of a C program source

save.s	indicates assembler source
ker_c5.lp	indicates output for line printer
note.m	indicates input for the mm processor

The following illustrate uses which do not respect the "FIRST PRINCIPLE":

progress.78	suffix indicates time the file relates to
names.all	suffix indicates extent of coverage of the file
status.new	suffix indicates time relevance of the file
old.src/bio.c	a period used in a directory name
alpha..beta	Poor taste??

1.2 When Suffixes

Nothing in this convention is meant to disallow the previously accepted practice of not using suffixes when they are not needed. It does however say that if you use one of the accepted suffixes all readers can assume you mean them to indicate what the convention suggests. Specifically, executable files and shells can be named without suffixes when they are in bin directories. Shells can be named without the ".sh" suffix in other directories if you insist, but marking them with ".sh" is considered a more "sociable" practice. (Some people follow the convention of naming shells with ".sh" in a working directory, but put it into their bin directory with a shell that places it under their own SCCS control in a source directory with the ".sh" intact and puts it in their bin directory with the ".sh" removed.)

Lots of other files exist without suffixes and without being executable. The convention does not ask that these be avoided, but that you avoid the use of period to separate parts of the name.

2. Conventional Suffixes

In the KSOS project we will adhere to the First Principles and corollaries mentioned in the previous section to decide when to use a file name suffix and how to indicate the suffix. This section will discuss accepted standard suffixes, which will be considered binding within the project, and some suggested standards which while not binding, are widely used and considered "socially acceptable".

2.1 Currently Binding Conventions

The suffix conventions in the following list are either currently enforced by a particular tool, or are eligible to be so enforced:

aj	Output ready for the Anderson-Jacobson terminal.

bak	Backups provided by the Rand editor or other backups.
bas	Input for the Unix Basic interpreter.
c	Source files for C programs.
f	Input for the Unix fortran compiler.
h	Header files for programs. Notice that this convention can be used regardless of whether the program is a C program or a MODULA program or an HDL design.
hdl	Input to the hdl processor. There are a large number of violations of this convention. They should be corrected at the first possible opportunity.
lx	Lex source grammar.
lp	Output ready for the line printer.
m	Source text files intended to be processed with mm.
m4	Input for the m4 macro processor. The convention is followed, but the processor is not used very often.
mac	Input for the macro-11 assembler.
mod	Input for the modula compiler.
o	Unix standard for output of compilers and assemblers.
p	Input for the proff or xroff processors. Most of the files using this convention are "old".
r	Input for the ratfor compiler.
s	Assembler source input.
sh	Shell files which reside somewhere besides in a bin directory.
sp	Source to be processed by the SPECIAL processor.
y yr	Yacc C-source grammar and yacc Ratfor source grammar respectively (prescribed by "make"").

2.2 Currently Non-Binding Conventions

These are conventions that people seem to be using extemporaneously. Where several things are being used for similar meanings a consolidation to one use is suggested.

a	Library or archive files. Typical use is represented by /usr/lib/libc.a.
bld	Input for the "build" tool.
fg	A figure ready for insertion into nroff with "so" command.
lock	A file which is created for the sole purpose of controlling exclusive use for some resource for a short period of time.
mk	Input to the PWB "make" tool.
out	Output which will probably be short-lived. For example, a.out, and other things of similar nature.
sed	Edit strings for the stream editor sed.
sk	Sketches using the terminal screen as crude graphics device or brief outlines of what one is about to write. Recommendation is that the suffix .sk mean you can look at it and fix it on your screen. After you process it through something else to make it ready to .so into an mm run the suffix should be changed to .fg. For example, if you use any backslashes in a sketch, what you want to look at on the screen is different from what mm will need. (Single occurance of backslash has to become four occurances when using the .DF command of mm.)
tbl	Input for the tbl macro processor.

The following are less often used, but hereby suggested:

eqn	Input for the eqn macro processor.
ed	Short edit sequences to be used with the y command of the D editor.
yc	Input for the yacc compiler-compiler.
[1-9]	In directories of manual pages this suffix indicates the section of the manual in which this manual page should reside. (Unfortunately, it appears that the manual was never intended to reach 9 sections.)
db	Data bases for use with the soon-to-be released data base tool being done by Glen Steedum.
sts	To replace all uses of stat, sta, status, etc., meaning input to a status monitoring tool.
x,xc,xf,xa	Files intended to be brought into a source file with the x, xc, xf, and xa commands of the d editor.

xh,xs,xm — Similar to the above, but until or unless the d editor is retrofitted, would have to be pulled in with x. They would be intended for use in hdl, special, or modula source files respectively.

3. Names in KSOS

Based on the commentary received in response to the draft of this set of conventions, the more acceptable practice seems to be to use capitalized CPC tags as directory names but to use lower-case for all other directories. The controlling reasoning seems to be that the capitalized tags are used in B5 special so that they will stand out as tags, and separating the other stuff to lowercase contributes to that "standing out".

The ollowing decisions are thus compromises.

3.1 Decision

With all the preceeding "good reasoning" to back it up, the following standard is offered:

THE SECOND PRINCIPLE OF FILE NAMING

> In the directory /kr/sunix (or whatever later replaces it) and all of its descendants, a directory name that is all capital letters is so named in order to indicate its applicability to the set of deliverable Computer Program Components and Computer Programs (CPCs and CPs) as specified in the KSOS index of CPCs (/kr/sunix/cpcindex/cpcindex).

To clarify the intent of this principle, we offer the following non-conflicting Corollaries:

Corollaries to the Second Principle

1. All fully-capitalized directory names will be an approved cpc tag.
2. All file names which apply to a CPC or CP will contain the proper tag in all uppercase characters. (It may of course contain other characters.)
3. A capitalized CPC tag used alone will indicate a directory; any other use of a CPC tag will have other characters with it, at least in a suffix (which is always lowercase).
4. Since directories are not managed by SCCS, a file name of the form s.<TAG>, without a suffix, will not be used.

It is our intent that these guidelines be followed scrupulously in the directories /kr/sunix/kernel, /kr/sunix/emulator, and /kr/sunix/nksr. We recognize that there would be very little value in going back to clean up old practices in directories other than these three.

3.2 Other Directories

We agreed earlier to use a directory structure that would allow specifications and hdl dealing with a particular level to exist in a specially named directory at that level.

Auxiliary Directory Names

doc	Documentation dealing with all the entities at this level.
hdl	HDL files dealing with the design at this level.
sp	Specifications (i.e. in SPECIAL) dealing with this level.
udf	Unit development folder. (By implication, this should appear only at the same level as a CP since unit development folders are by CP.)

In addition, some other practices seem to have come into being which have not yet been discussed by the whole group. We offer them here as recommendations.

gen	A directory full of shells to generate the stuff in its parent directory.
inspect	Similar to gen above, but with precise intent to run off everything that will be needed for the design/code inspection.
sketch	A directory of sketches to support a document whose main body is in the parent directory.
sp	A directory of files which contain the Special source for this CPC or CP.
c5_pieces	A directory of stuff to assist in generating the C5 prose for a given level, or in fact the C5 prose itself. Note that the special for this level is expected not to be hidden here, but instead should be in a directory "sp".
???	A number of directories seem to be CPC names that are not in the cpcindex. Note that the cpcindex is constructed from /kr/sunix/sked and that if you haven't fixed it to correspond to your current list of index tags all bets are off.

4. Review of Directory Structure

In spite of the fact that nobody wrote down or otherwise saved a reliably recoverable statement of the directory structure to be used for KSOS, I have talked with a number of people about their recollections and current practices. The following is meant to be an accurate representation of that agreement.

4.1 Basic Structure

The KSOS files are kept in the directories /kr/sunix/kernel, /kr/sunix/emulator, and /kr/sunix/nksr for the three CPCIs which comprise this project. Any files dealing with all three of the CPCIs are hung in directories at this same level, for example /kr/sunix/sked contains files dealing with overall project scheduling.

Within the CPCI directory are files dealing with the entire CPCI, directories of prose, hdl, and/or special which are concerned with the entire CPCI, and directories whose names are capitalized CPC tags (lets call them CPC directories.)

Within the CPC directories are files dealing with the entire CPC, directories of prose, hdl, and/or special which are concerned with the entire CPC, and directories whose names are capitalized CP tags (lets call them CP directories.)

Within the CP directories are files dealing with the single CP and directories of prose, hdl, notes, shells, and/or special which are concerned solely with this CP.

It is considered acceptable, for example, to not have a subordinate directory to contain the hdl for a given CP if it is just as easy to keep the hdl in a small number (1-5) of files at this level marked by the suffix ".hdl". However, it is considered better to have any files that deal with the hdl (for example a shell to run it all out to the printer) visible from the same ls command that will show the hdl files.

4.2 Examples

The following have been extracted from existing practice and modified as needed to illustrate "accepted practice".

```
emulator/EI/EBC/doc/s.EBC_c5.m
emulator/EI/EDV/DV_disp.hdl
emulator/EI/EDV/doc/s.EDV_c5.m
emulator/EI/EFI/disp_1.hdl
emulator/EI/EFI/doc/s.EFI_c5.m
emulator/EI/EFI/hdl/p.EFI.hdl
kernel/KIO/IOS/s.IOSecure.hdl
kernel/KIO/IOS/s.IOSecure.m
kernel/KIO/IOT/s.IOTerm.hdl
kernel/KIO/IOT/s.IOTerm.m
kernel/KIO/IOU/s.IOUtil.hdl
kernel/KIO/IOU/s.IOUtil.m
nksr/NSO/ASG/s.ASG_c5.m
nksr/NSO/DAS/s.DAS.hdl
nksr/NSO/DAS/s.DAS_c5.m
nksr/NSO/LPS/p.LPSl.hdl
```

5. Other Principles

These are general preferences and suggestions which for the moment are not binding. Readers and others concerned are invited to participate in discussions/dialogue about the advisability of any of these, and to offer their own principles to add to the list.

1. A name of a shell should begin with the abbreviation of the nearest applicable "verb". For example, a shell which principally does a grep should be named "grp_<something>.sh". Other abbreviations and the verbs which they imply are : gen-generate, hdl-run_hdl_on_it, spc-run_special_on_it, fnd-find, drv-derive_something_from_it, prt-print. Suggest your own favorites. Note that in short names like this it is better to put the verb abbreviation first, because some things could be either verbs or nouns. For example, prt_hdl.sh means a shell that prints hdl output.

2. The underscore should be used to separate abbreviations in a name when that is feasible and needed for clarity.

3. File names which are not executable and not shells should be constructed so as to avoid confusion with those which are. This means avoid an obvious abbreviation for a verb as first part of a name. For example, don't use a file named run_test to mean the output of a test run. A better name for this would be test_rslt.lp or something similar.

6. Implementation Targets

We'll target for compliance within the three directories of immediate interest by the end of the day Friday.

After that time any new conventions that we think we'd like to have we'll simply have to live without. That is, we can't take the time to do any further reworking of file names without adversely affecting progress.

LANGUAGE

SPECIAL Language Specification Standards for KSOS

Lawrence Robinson

Ford Aerospace and Communications Corporation

SPECIAL (SPECIfication and Assertion Language) is the formal specification language of HDM (Hierarchical Development Methodology). HDM provides a model of computation that consists of a structuring of system components. SPECIAL is the language for describing the external behavior of the components. (*)

The format and style of SPECIAL specifications are extremely important, perhaps even more so than for programs, for the following reasons:

- Formal specifications serve as documentation for the system and will be extensively read.
- Because of the differences between SPECIAL and programming languages, SPECIAL specifications are difficult for inexperienced people to read and understand, even when clearly written.
- A formal specification language such as SPECIAL has much power. The power of SPECIAL, in particular, makes it extremely easy to write specifications that are impossible to read and understand.
- Verification requirements (especially those for design proofs) set many restrictions on specification style that cannot be enforced by purely syntactic means.
- As distinguished from programs, there is no excuse, such as efficiency, for poor style, because formal specifications are not be executed.

This document contains guidelines and restrictions concerning the style and format of SPECIAL specifications imposed by readability considerations, good specification practice, and verifiability.

In HDM the unit of specification is a module. Each module contains

1. A set of internal data structures that describe the state of the module.
2. A set of operations that allow a calling program to access or modify the internal data structures of the module. The internal data structures cannot be accessed or modified in any other way.

A SPECIAL specification contains

1. A declaration of the internal data structures of the module as primitive V-functions or VFUNs. The initial values of the primitive VFUNs must be supplied.

---

* SPECIAL is used to describe the mappings or relationships among the data in the system components, but since there are no mappings in the KSOS specifications at this time, the treatment of mappings is outside the scope of this document.

2. A specification of each operation. The three types of operations are: visible VFUNs (which return values but do not change the state); O-functions or OFUNs (which change the state but do not return values; and OV-functions or OVFUNs (which both change the state and return values). The returned values are expressed in terms of expressions containing the primitive VFUNs of the module. The state changes are expressed as assertions (BOOLEAN-valued expressions) that describe the relation between the values of the internal data structures before the operation's execution (unquoted) and the values of the internal data structures after the operation's execution (quoted).

For more information on SPECIAL and HDM, consult

- Karl N. Levitt, Lawrence Robinson, and Brad A. Silverberg, "The HDM Handbook, Volume III: A Detailed Example in the Use of HDM," Technical Report, SRI International, Menlo Park, CA, June 1979.
- Lawrence Robinson, "The HDM Handbook, Volume I: The Foundations of HDM," Technical Report, SRI International, Menlo Park, CA, June 1979.
- Olivier Roubine and Lawrence Robinson, "SPECIAL (SPECIfication and Assertion Language): Reference Manual," TR-CSG-45, SRI International, Menlo Park, CA, January 1977.
- Brad A. Silverberg, Lawrence Robinson, and Karl N. Levitt, "The HDM Handbook, Volume II: The Languages and Tools of HDM," Technical Report, SRI International, Menlo Park, CA, June 1979.

1. Module Format

This section is concerned with the lexical layout of SPECIAL specifications and the form and content of comments.

A module specification in SPECIAL is organized into the following sequence of paragraphs, each of which may or may not be present: TYPES, DECLARATIONS, PARAMETERS, DEFINITIONS, EXTERNALREFS, ASSERTIONS, and FUNCTIONS. Each paragraph contains the definition of one or more objects. Objects that are functions (which specify the internal data structures and operations of a module) are divided into several optional sections.

1.1 Comments

The use of comments in SPECIAL is extremely important, because it is often the comments alone that make the difference between a readable and an unreadable specification. Comments should be used as liberally as possible; it is better to err in the direction of too many comments than too few.

Comments in SPECIAL are delimited by a distinguished set of brackets:

$( )

It is not permitted to have any of the following characters

$ ( ) [ ]

in a comment, without enclosing the offending parts of the comment in double quotation marks. Double quotation marks may occur within comments, but only in pairs.

Every module specification should have an introductory comment (immediately following the tokens "MODULE modulename") explaining the basic workings of the module and the unique global techniques used to specify the module's functionality. The following questions should be answered:

- What does the module do?
- Are there any particular restrictions or assumptions concerning the use of the module?
- What kind of data does it manipulate?
- How is its internal data structured in terms of VFUNs?
- What assumptions does it make about other modules?
- What are the useful properties of the data?
- What classes of operations does it perform?

The introductory comments should give the first-time reader a good idea of what he has to know to read and understand the formal specifications to follow.

The declaration of each object of the module specification should be commented, explaining what the purpose of the object is. Of course, mnemonic names should help, but are usually no substitute for a good explanation.

An externally-referenced object need not be commented, except to state why that object is externally referenced. The comments concerning types and parameters need not be long, except to state what the objects mean and why they are there. Comments for these objects should appear immediately following the declarations of these objects.

Comments for the specifications of (global) definitions, (global) assertions, and functions should be longer, because these objects contain expressions that must be understood. If any of these kind of expressions performs a nonstandard manipulation to achieve its goal, this must be explained carefully.

The comment for each global definition should state what the definition does and why it must be used. The comment of the definition should specifically state the relationship of the arguments to the result of the definition. A general comment must appear at the end of a definition. Comments at interesting places within the definition may also be inserted.

The comment for each global assertion should intuitively state the essential property guaranteed by the assertion (this is almost always different from its embodiment in SPECIAL) and why this property is important. A general

comment must appear at the end of an assertion. Comments at interesting places within the assertion may also be inserted.

The comment for the specification of a function should appear immediately following the function header. The comment should state what the function does, the importance of its arguments and result, and a brief summary of the assumptions and exception conditions. Comments at interesting places within the function's specification may also be inserted.

1.2 Indentation

The title of each paragraph should be indented four spaces. Each object declaration should begin on a new line, preferably at the beginning (except if the object is a type or a parameter declared in a list, in which case it may be indented). In external references, the "FROM" part should be indented four spaces. The name of each section within a function should be indented two spaces. Each line of the body of each section should be indented at least four spaces.

Indentation at the expression level should be in keeping with the nesting level of the operators in the expression. The following guidelines should be observed:

1. An expression should be made to fit on a line if possible.

2. If an expression must be broken, it should be broken at the outermost nesting level possible.

3. The following schemas of breaking according to nesting level or block structure are acceptable:

   Infix operators

```
  x
OP y
OP z

x
  OP y
  OP z

x    OP
  y  OP
  z
```

   Functional operators

```
f(a,
  b,
  c)

f(a, b,
  c)
```

Block-structured operators

```
IF a
  THEN b
  ELSE c

FORALL xtype x
  | p(x)
  : q(x)

FORALL xtype x :
  p(x)
  q(x)
```

4. Declarations of formal arguments or the fields of structures are ordinarily put as many on a line as possible, but may be allocated one to a line and indented the same amount of space as the others (especially if each argument or field is commented).

5. Within a function specification, each definition, exception, or effect should begin on a new line.

1.3 White Space

White space, both horizontal and vertical, should be used to increase readability. The following are minimal guidelines for white space, but more may be inserted if desired.

For horizontal white space, at least one space (or a new line) should surround all infix operators, except brackets,

```
( ) [ ] { }
```

separators and terminators,

```
; ,
```

and miscellaneous operators.

```
' .
```

Any reference to a function or an elements of a vector should contain no space between the name of the function or vector and the left parenthesis or square bracket. There need not be spaces between the brackets and the expressions within the brackets. Separators must have a space after the separator, but no space before it. A single quote must be preceded by a space, but no space should follow it. A dot (i.e., a structure selector) should have no space on either side of it.

As far as vertical white space is concerned, there should be at least two blank lines immediately preceding each paragraph name and at least one following it. At least one blank line should separate the declarations of global definitions, groups of external references from a single module, global assertions,

and functions (not externally referenced). There should be at least one blank line between the last line of the last function and the "END_MODULE."

2. Specification-Level Structure

This section is concerned with the structure of a SPECIAL specification at the gross, or specification level.

There should be no more than one module specification per file. Care should be taken concerning the size of the module specification, because all existing SPECIAL processors (both at FACC and SRI) have limitations in the size and complexity of the module specifications that they can process, which is usually a complex function of the length of the specification and the number of symbols declared in it.

2.1 Absence or Presence of Paragraphs

The DECLARATIONS paragraph is not to be used, meaning that "global" declarations of variables within a module specification are not permitted here, although both SPECIAL processors allow them. This regulation has been applied for reasons of style and readability.

Each of the other paragraphs is optional, as stated above.

2.2 Orderings

There is a partial ordering of external references among modules. This is not enforced by the FACC SPECIAL processor, but is necessary for compliance with the rules of HDM.

The order of the paragraphs within a module specification is fixed, as stated above. Within each of the paragraphs there is an ordering as well, corresponding to the dependence among the definitions of the objects in the paragraphs.

In the TYPES paragraph, the ordering of the type declarations is as follows:

- Those types declared in externally referenced modules, in the order that the modules are externally referenced.
- The types first declared in the current module.

Within the set of types for each module, the order of declaration should proceed from the "most primitive" type to the "least primitive" type. Thus, any type definition T1 containing a reference to a type definition T2 should follow the definition of T2 in the module specification.

As with the types, the global definitions are declared such that any definition D1 referring in its body to a definition D2 should follow D2 lexically in the module specification.

As stated above, the external referencing among modules forms a partial ordering. The modules externally referenced in the specification of a given module M should be listed in the specification of M according to the partial ordering. Thus, if the specification of M references modules M1 and M2, and if the specification of M1 references M2, then the reference to M2 in M should follow the reference to M1 in M.

2.3 Consistency

The SPECIAL processors at FACC and SRI perform strong type checking on a single module. However, only the SRI SPECIAL processor performs intermodule type checking. Since it is necessary for a set of module specifications to be consistent in the referencing of types, this checking must be performed by hand if an adequate online tool is not available.

There is one further restriction on intermodule type referencing that is not performed by any type checker. All non-exportable types (e.g., sets, vectors, structures, and subranges) must be redeclared in modules that need such types in their external references. Currently, these types need not have the same names or even the same definitions (e.g., structure names, exact subrange identity) to pass the type checking. In KSOS, all externally referenced types, whether exportable (e.g. designator types, scalar types) or non-exportable, must be identical in both name and definition.

The SRI SPECIAL processor forbids having two structures in the same module with intersection selector name sets; It also forbids having a structure selector clash with a variable name. These restrictions, although not enforced by the FACC SPECIAL processor, apply to the KSOS formal specifications.

It is also forbidden to have a variable name at an inner scope (e.g., a quantifier) be the same as any name in an outer scope.

2.4 Further Restrictions

All types used outside the TYPES or EXTERNALREFS section of a specification must be named types. Thus, a type name may be defined in the TYPES section, e.g.,

```
x: VECTOR_OF CHAR;
```

and then used in the declaration of a variable, e.g.,

```
OFUN f(x string);
```

However, the type cannot be used in-line, e.g.,

```
OFUN f(VECTOR_OF CHAR string);
```

This is not a current rule in SPECIAL, but failure to comply often causes difficulty, especially with structures.

All "infinite" or mathematical types, when used as arguments or results in operations, should be limited to finite subranges or sizes. For example,

```
x: INTEGER;
y: VECTOR_OF CHAR;
```

should be changed to

```
x: {0 .. 2^16 - 1};
y: {VECTOR_OF CHAR vc | LENGTH(vc) <= max_string};
```

This practice is in keeping with the use of the type "cardinal" in the KSOS programming languages.

As stated above, there is to be no DECLARATIONS paragraph in any formal specification for KSOS, and thus there are no globally declared variables.

The use of sets as arguments and results in operations is prohibited, with one exception. This prohibition arises from the fact that the KSOS implementation languages do not support a SET type. However, a set of flags (in a scalar or enumerated type) may be declared in a formal specification, because this can be converted to a BOOLEAN array indexed by the enumerated type in Modula, and to a bit vector in C.

The use of the type REAL, and all operations on REAL quantities, are prohibited in KSOS SPECIAL specifications.

The version of SPECIAL used in KSOS and accepted by FACC's processor does not include union types (i.e., ONE_OF) or the TYPECASE construct.

Likewise, parametric functions, e.g.,

```
        PARAMETERS

    INTEGER factorial(INTEGER x);
```

are also prohibited from the subset of SPECIAL to be used in KSOS. They are not accepted by FACC's SPECIAL processor.

2.5 Stylistic Considerations

In most software engineering methods there is a tendency to emphasize small computational units, in this case HDM modules. However, this is a goal in HDM that must be traded off against the number of external references (which begin to proliferate when the module size gets too small).

In systems such as KSOS, where a module's state contains a lot of different information, there is the issue of how to structure the state in terms of VFUNs. Often, when there are parts of the state that are closely coupled (i.e., referenced by the same arguments, accessed and changed at the same times), it is useful to combine such data into a structure that is returned by a primitive VFUN of the desired arguments.

Definitions, both local and global, should be used extensively in the KSOS formal specifications to provide a formal development for the essential system concepts and to make these concepts more understandable.

In functions, all exception conditions must be named (this is not a requirement in SPECIAL). Exceptions that contain security checks should occur as early in the list of exceptions as possible; this reduces the complexity of the formulae generated for the design proofs.

3. Assertion-Level Issues

This section is concerned with the structure of a SPECIAL specification at the detailed, or assertion level (i.e., the level of an individual arithmetic or BOOLEAN expression).

3.1 Stylistic Considerations

In the writing complex expressions that are parts of effects and definitions, sets and quantifiers were originally the constructs of choice. Because of a desire to increase both readability and compatibility with the Boyer-Moore theorem prover, formulating these complex ideas using recursive definitions is now preferred. These definitions should be used extensively to reduce the size and redundancy — and to increase the readability — of complex expressions written in SPECIAL.

SPECIAL has many powerful constructs (e.g., definitions, quantifiers, the LET and SOME constructs) that can be composed in arbitrary ways. Extremely complex expressions can be made in this way. This adversely affects the verifiability, as well as the readability of the formal specifications, and should be minimized at all cost.

3.2 Restrictions

The use of the constructs EXCEPTIONS_OF and EFFECTS_OF should be minimized, if not completely done away with. The original intention of these constructs was to save a lot of repetition for the writer of a formal specification. However, these constructs have been overused, to the point where they

- Have reduced some formal specifications to an extremely long chain of these constructs, corresponding exactly to the procedure call hierarchy of the implementation.
- Caused difficulty in generating the formulae for the design proofs.
- Can be applied with a more procedural semantics in mind, resulting in specifications that are logically inconsistent.

The semantics of these constructs is that of an exact, inline expansion. To help clear up the above problem, the following restrictions have been devised:

- These constructs cannot be used to refer to operations defined within the same module. There are other means (e.g., definitions) for avoiding repetition within the same module specification.
- EFFECTS_OF can be used only when the set of V-function positions (*) whose

values are changed by the construct does not intersect with the set of V-function positions whose values are changed anywhere else in the effects of the function where the construct occurs (this could mean another EFFECTS_OF, including one created by another instance of the same quantified variable). The EFFECTS_OF construct was not originally intended to be used within quantifiers, SOME, or LET. Great care must be taken in these cases.

- EFFECTS_OF may not be used within a definition. This often requires, when referring to the results of an OVFUN, having — in addition to the specification of the OVFUN -- the specification of a VFUN that returns what the OVFUN would have returned had it been executed.

The constant UNDEFINED (or ?) is not to be represented in any way in the implementation. Having the value of a primitive VFUN, for certain arguments, be UNDEFINED means that this value is not being supported by the module in the present state. A visible VFUN must never return UNDEFINED; any call to a visible VFUN must trigger an exception if it would have returned UNDEFINED.

As stated above, the TYPECASE expression is forbidden.

Whenever possible, state changes in KSOS formal specifications should be of the form

```
. . .
'f(args) = expression of old state
. . .
```

The major case where such a restriction cannot be enforced is in the case of non-determinism. However, these cases can be minimized if care is taken.

It is the policy in SPECIAL that if any V-function position is not mentioned in the EFFECTS section of the specification of an operation (invoked in a certain state, with a certain set of argument values), it is considered not to have changed as a result of invoking the operation. However, this policy does not extend to a granularity finer than that of the V-function position. Thus, if one wishes to specify a change to a single field of a structure (that is the value of a V-function position) or a single element of a vector (that is the value of a V-function position), then one must specify the new value of the entire structure or vector.

---

* A V-function position is a V-function name and a single tuple of argument values, denoting a "location" that can have a value.

HDL Standards for KSOS

This paper outlines HDL Standards for KSOS. The standards were adopted to achieve a high degree of precision and commonality across different designs.

These standards give the term module a denotation that differs from its denotation in the HDL manual. The term module shall denote a Modula like module and the term procedure shall denote a procedure or a function.

1. Design Heading

Each design of a CPC (Computer Program Component) shall have a header that contains the following information:

CPCI - The Computer Program Configuration Item of which this design is a part.

CPC - The Computer Program Component that this design describes.

TAG - A KSOS system wide unique tag given to all objects exported from this CPC.

MODULE NAME - The name given to this CPC.

A Specs - The section in the A Specs that this design describes.

B5 Specs - The section in the B5 Specs that this design describes.

PRIVILEGES - The security privileges that this CPC has.
A "d" editor "x" prompt file (/usr/sunix/hdl/hdr) for the described module header exists.

2. Procedure Headings

All HDL procedures must have a heading. This heading should appear within comments. ie.

```
/*----------------------------------------------------------------

        Procedure Heading

----------------------------------------------------------------*/
```

The procedure heading should contain the following information.

PURPOSE - The "purpose" statement should be a concise description of the purpose of the procedure. The syntax and semantics of this statement is given in the HDL manual. This statement in mandatory in all procedures.

DESCRIPTION - The "description" statement is a high level description of the workings of the procedure. This description can be in english or HDL. The syntax for this statement should be the same as the "purpose" statement. The syntax of the "description" statement is:

```
<description statement> ::=
        description  { <description list> }

<description list> ::=
        <empty>  |
        <description list>  <word_list> ;
```

The definition of "word_list" is given in the HDL manual.

ASSERT - The "assert" statement makes explicit the conditions that must be true in order for the procedure to operate correctly. A full discussion of the "assert" statement appears in the exception handling section.

EFFECT - The "effect" statement describes all the global changes made by this procedure. The syntax and semantics of this statement is given in the HDL manual.

IMPORT - The "import" statement allows a procedure or module to access objects that exist in an external scope. The "import" statement shall list all the external objects that are referenced by a procedure or module. The import statement has the following syntax.

```
<import> ::=
        import  {  <import list>  }

<import list> ::=
        <empty>  |
        <import list>  <import item>


<import item> ::=
        <identifier list>  :  <class>  ;


<class> ::=
        <constant>  |
        <type>      |
        <variable>  |
        <procedure>


<constant> ::=
        constant


<type> ::=
        type
```

```
<variable> ::=
        read | read_write

<procedure> ::=
        procedure
```

For example:

```
import
{
        object_measurements : type;

        height, volume  : read;
        name            : read_write;

        surface_area    : procedure;
}
```

EXPORT - The "export" statement allows a module to give access permissions to the external environment of objects that exist within its scope. The notion of module is new to HDL. A module to be a collection of procedures and data, where no object internal to the module is known to the external environment, unless it is exported. With the exception of the key word "export", the syntax for the "export" statement is the same as the "import" statement. The "export" statement has the following syntax:

```
<export> ::=
        <export>  {  <export list>  }

<export list> ::=
        <empty>  |
        <export list>  <export item>

<export item> ::=
        <identifier list>  :  <class>  ;
```

For example:

```
export
{
        object_measurements : type;

        height, volume  : read;
        name            : read_write;

        surface_area    : procedure;
}
```

EXCEPTIONS - The "exceptions" statement states the how different exception conditions are handled. A full discussion of the "exceptions" statement appears in the exception handling section.

CONSTANT - This section is used to declare constants. The syntax of the constant declaration is:

```
<constant declaration> ::=
        constant  {  <constant list>  }

<constant list>  ::=
        <empty>  |
        <constant list> <identifier> := <constant expression>
```

Constant expression is a constant or an expression involving only constants.

TYPE - This section is used to declare types. The syntax for type declaration section is:

```
<type definition section> ::=
        type  {  <type definition list>  }

<type definition list> ::=
        <empty>  |
        <type definition list> <type definition>
```

Discussion of the syntax and semantics of the type definitions deferred to the section on types.

VARIABLE - This section is used to declare variables. The syntax for a variable declaration part is:

```
<variable declaration part> ::=
        variable  {  <variable declaration list>  }

<variable declaration list> ::=
        <empty>  |
        <variable declaration list>  <variable declaration>
```

Discussion of variable declarations is deferred to the section on variable declarations.

3. Type Definitions

Type definitions are placed with constant and variable declarations in module and procedure headings. A type is the set of permissible values which a variable of that type may assume. Types are divided into three primary categories: simple types, structured types, and pointer types.

3.1 Simple Types

3.1.1 Enumerated Type — The constants of the type are explicitly enumerated by the designer. These constants obey the same formation rules as other identifiers. An example of an enumerated type definition is:

```
day = (Sun,Mon,Tue,Wed,Thu,Fri,Sat);
```

The constants "Sun", "Mon", etc. are the actual literal values of the type day, just as "1", "2", etc. are actual literal values of type integer. The constants of an enumerated type are regarded as being ordered, or scalar. For instance, it would be possible, though tedious, to enumerate the values permissible as the type "integer" in a particular implementation.

3.1.2 Subrange Type — A subrange of another scalar type may be defined as a new type. Examples of subrange types are:

```
workday = monday .. friday;
oneto10 = 1 .. 10;
```

The subrange type inherits the operators of the scalar type of which it is a subrange. However, the subtype is not necessarily closed with respect to these operators.

3.1.3 Type Identifiers — Any type identifier is a simple type. This may be a user defined type identifier or an HDL standard type. The HDL standard types are: integer, real, boolean, char, string, and object.

3.2 Quantified Type

Any subset of an existing type derivable by a quantification of the form "all t : T1 such that P(t)", may be defined as a new type. Here the scope of the identifier "t" is this expression only, "T1" is a simple type, and P is a predicate expression. In,

```
t2 = all t : T1 such that P(t);
```

the type T2 is a subset of the type T1.

3.3 Structured Types

3.3.1 Array Type — This structured type consists of a fixed number of components all of the same type known as the component type. The array is indexed by the values of another simple, scalar type known as the index type. Several index types may be specified to yield a multidimensional array. Examples are:

```
sq10matrix = array [oneto10,oneto10] of real;
bitvector = array [0..15] of boolean;
translateTab = array [char] of char;
tvec = array [-19..32] of 50..75;
```

Note that is also possible to have arrays of arrays as in:

```
bitmap = array [0..127] of bitvector;
```

In this case each element of an array of type bitmap is itself an array of type bitvector. If variable v is of type bitmap then a reference to an individual bit (boolean) is of the form v[i][j], whereas a reference to a bitvector in the bitmap is of the form v[i].

3.3.2 Struct Type — These types are cartesian products of other types. For example :

```
complex = struct {
              re, im : real;
          };

SEID = struct {
           NSP : 0..255;
           UID : 0..16777215;
       };
```

If a variable z is declared to be of type complex, the real and imaginary fields are referenced as z.re and z.im respectively.

In addition to simple cartesian products, there are discriminated and nondiscriminated unions. Struct types actually consist of a fixed part and a variant part either of which may be empty. The example structures above have nonempty fixed parts but empty variant parts. A structure with a nonempty variant part is a union (discriminated or nondiscriminated). The variant part of a structure defines several alternative field lists only one of which is used, at a particular instant in time, to describe the structure of the variant part of a particular instantiation of the struct.

In a union, a distinguished field may be specified, which effectively exists in the fixed part of the struct, and is assigned a value which identifies the particular variant. Such a structure is then called a discriminated union, since it contains this distinguished field, or tag field, to discriminate among possible variants. There exits a reserved label "default", such that if the distinguished field contains a value that is not listed as one of the possible variants, then the "default" variant of the union is accessible. A structure not declaring a tag field is called a nondiscriminated union. The variant to be used in interpreting the data of such a structure must then be determined by flow or storage states outside of the structure. An example of a discriminated union is:

```
person = struct {
             name : string;
             age : integer;
             union ( sex : (male,female) ) {
                 male:
                     {
                         mustachioed : boolean;
                         courageous : boolean;
                     }
                 female:
                     {
                         pretty : boolean;
                         sizes : array [1..3] of integer;
                     }
             }
         };
```

This record might be used in the following way for a variable Harry of type person:

```
Harry.name := "Harcourt Fenton Mudd";
Harry.age := 41;
Harry.sex := male;
Harry.mustachioed := TRUE;
Harry.courageous := FALSE;
```

Because male was assigned to the sex field, a reference to the field Harry.pretty, for example, would be an error.

It is possible for the variant associated with a particular value of the tag field to be empty, and thus be represented as simply "{ }". An example of a nondiscriminated union is:

```
psw = struct {
          union {
              {   integer i;   }
              {   /* word packed from low bit to high */
                  C_bit, V_bit, Z_bit, N_bit, T_bit : boolean;
                  priority : 0..7;
                  fill : 0..15;
                  previous_mode, current_mode : 0..3;
              }
          }
      };
```

3.3.3 Set Type — A new type may be defined which is the powerset (i.e. set of all subsets) of some base set. The base set may be any simple type. An example of a set type definition is:

```
fileDescriptor = 0..15;
fileSet = set of fileDescriptor;
```

then,

```
VARIABLE {
    openfiles, inputfiles, outputfiles : fileSet;
}
```

declares three variables to represent sets of file descriptors.

3.4 Pointer Types

A pointer type is a set of values which may be used to designate variables of another specific type to which the pointer type is bound. The distinguished standard value NIL belongs to every pointer type. The value for a pointer to a particular variable is returned by a call on the generic standard function address(). The function invocation address(v) for a variable v of type T yields a pointer value of type ^T.

3.5 Syntax of Type Definitions

Type definitions occur in module and procedure headers in the form of a type definition section. The grammar for type definitions follows:

<type definition> ::= <identifier> = <type> ;

<type> ::= <simple type> | <quantified type> |
<structured type> | <pointer type>

<simple type> ::= <enumerated type> | <subrange type> | <type identifier>

<enumerated type> ::= (<identifier list>)

<identifier list> ::= <identifier> | <identifier list> , <identifier>

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<quantified type> ::=
all <identifier> : <simple type> such that <predicate>

<predicate> ::= <expression>

<structured type> ::= <array type> | <struct type> | <set type>

<array type> ::= array [<index type list>] of <component type>

<index type list> ::= <index type> | <index type list> , <index type>

<index type> ::= <simple type>

<component type> ::= <simple type>

<struct type> ::= struct { <field list> }

<field list> ::= <fixed part> <variant part>

<fixed part> ::= <empty> | <field definition list>

<field definition list> ::=
<field definition> | <field definition list> <field definition>

<field definition> ::= <identifier list> : <type>

<variant part> ::=
<empty> | union <opt tag field> { <variant definition list> }

<opt tag field> ::= <empty> | (<identifier> : <simple type>)

<variant definition list> ::=
<variant definition> | <variant definition list> <variant definition>

<variant definition> ::= <opt labels> { <field list> }

<opt labels> ::= <empty> | default : | <label list> :

<label list> ::= <constant> | <label list> , <constant>

<set type> ::= set of <base type>

<base type> ::= <simple type>

<pointer type> ::= ^ <type identifier>

4. Variable Declarations

Variable declarations are placed with constant and type declarations in module and procedure headings.

4.1 Syntax of Variable Declarations

The grammar for variable declarations follows:

<variable declaration> ::= <identifier list> : <type>

<identifier list> ::= <identifier> | <identifier list> , <identifier>

5. Set Expressions

Expressions of type "set of <base type>" appear in the statement part of an HDL procedure or function. They may be assigned to set variables of the same type or appear in set relational expressions.

5.1 Syntax of Set Expressions

<set expression> ::=
<set function reference> | <set variable> | <constructed set>

<set function reference> ::= <function reference>

<set variable> ::= <variable>

<constructed set> ::= [<set constructor list>] | [ ]

<set constructor list> ::=
        <set constructor> | <set constructor list> , <set constructor>

<set constructor> ::=
        <element> | <element> .. <element> |
        all <identifier> : <universe set> such that <predicate>

<element> ::= <base type expression>

<universe set> ::= <simple type> | <set expression>

<set constructor> ::= <expression> | <expression> .. <expression> |
        all <identifier> : <simple type> such that <predicate>

Additionally, set expressions may be used in relational expressions as defined by the following partial grammar for relational expressions:

<relational expression (set)> ::=
        <set expression> <set relation operator> <set expression> |
        <element> IN <set expression>

<set relation operator> ::= < | > | <= | >=

5.2 Examples of Set Expressions

The following examples declare set types and set variables, and demonstrate the construction of valid set expressions in statements.

EXAMPLE 1.

```
    TYPE {
        day = (sun,mon,tue,wed,thu,fri,sat);
    }

    VARIABLE {
        today : day;
        weekdays, weekends : set of day;
    }

    {
        weekdays := [ mon .. fri ];
        weekends := [ sat, sun ];

        today := getday();

        if( today in weekdays )
```

```
            get up;
        else
            go back to sleep;
    }
```

EXAMPLE 2.

```
    TYPE {
        intset = set of integer;
    }

    VARIABLE {
        even, odd : intset;
        positive, negative, natural : intset;
        primes : intset;
    }

    {
        even := [ all i:integer such that i%2 == 0 ];
        odd  := diff( integer, even );

        positive := [ all n:integer such that n >= 0 ];
        negative := [ all n:integer such that n < 0 ];
        natural  := diff( positive, [0] );
        primes   := [ all k:positive such that is_prime(k) ];
    }
```

EXAMPLE 3.

```
    TYPE {
        oneto20 = 1..20;
        set1to20 = set of oneto20;
    }

    VARIABLE {
        s1, s2, s3, s4 : set1to20;
        b : boolean;
        i : integer;
    }

    {
        s1 := [2,5,9,10,17];
        s2 := [1,2,11..15,19];
        b  := s1 < s2;
        s3 := intersection(s1,s2,[all e:set1to20 such that e<5 .and. 9<e]);
        read(i);
        if( i IN [-19..19] )
                s4 := [abs(i)+1,1,20];
    }
```

6. Operators

The following list of operators are defined. The operators in the same groups have the same priority. Expressions with operators of the same priority are grouped left to right.

All operators assume that both operands are of the same type.

Multiplicitive Operators

```
*    real    * real     -> multiplication
     integer * integer  -> multiplication

/    real    / real     -> Division
     integer / integer  -> Division

%    integer % integer  -> Modulus
```

Additive Operators

```
+    real    + real     -> Addition
     integer + integer  -> Addition

-    integer - integer  -> Subtraction
     real    - real     -> Subtraction
```

Comparision Operators

The comparison operators return a boolean value dependent on the outcome of the comparison. The operands of all comparison operators must be of the same type, with the exception of the "in" operator.

```
==   All types     -> Equal.

~=   All types     -> Not equal.

<=   Ordered type  -> less than or equal to.
     Set type      -> Subset.

>=   Ordered type  -> Greater than equal to.
     Set type      -> Superset.

>    Ordered type  -> Greater than.
     Set type      -> Proper superset.

<    Ordered type  -> Less than.
     Set type      -> Proper subset.

=>   Boolean       -> Implication.

in   element in set -> Set inclusion.
```

Logical Operators

The logical operators take only boolean operands and produce a boolean result.

~ Not.

|| Or.

&& And.

7. Intrinsic Functions

The following is a list of functions that are understood to be defined.

Allocation

new(type) - This function returns a pointer to a new area that can only contain the given type.

Sets

cardinality(set) - This function returns the number of elements in the given set.

ordinal (set elememt, type identifier) - This function returns the ordinal position in the scalar type denoted by type identifier. If the type identifier is absent, this function returns the ordinal position of this element in the largest scalar type in which it is defined.

union (set1, ... setm) - The union of the sets "set1" through "setm" is returned. Note that all the sets must be of the same type.

intersetion (set1, ... setm) - The intersection of the sets "set1" through "setm" is returned. Note that all the sets must be of the same type.

Type Information

low (scalar type) - The lowest value allowed in this type is returned.

high (scalar type) - The highest value allowed in this type is returned.

Type Conversion

Since the operators in HDL require their operands be of a specific type, it is necessary to do type conversions to the operands before applying the operator. Type conversion functions can also be used to construct constants of the desired type. For example, if the type complex is declared as:

```
complex = struct { re, im : real };
```

and "x" is of type complex, we can use a complex type conversion function to create a complex constant to test the value of "x".

```
if ( x == complex (3.5, 1.2) )
```

In the following functions only one of the arguments in square braces may appear in the function call.

integer([real,char,string])

real - The real number is truncated and an integer value is returned.

char - The integer representation of the character is returned.

string - The integer value represented by the string is returned. Note that this value is truncated if necessary.

real([integer,char,string])

integer - The real representation of the integer is returned.

char - The real representation of the value of the character is returned.

string - The real value represented by the string is returned.

string([real,integer,char])

real, integer - The string representation of the numeric value is returned.

char - The string containing that single character is returned.

8. Style

8.1 Constants

When possible, designers should avoid using strings and numeric constants within the body of procedures. Constant identifiers should be used in their place. A constant identifier is declared in the constant declaration part of the procedure heading. For example:

```
CONSTANT
{
        BUFFER_SIZE  := 128;
        MY_DIRECTORY := "/u/my_directory";
}
```

8.2 Implicit Assignments

This suggestion is an extension to HDL. Understanding where and when variables are given new values is necessary for the comprehension of the design. Passing an argument by reference is an implicit assignment. This pass by reference can be made explicit if all parameters of a procedure call are assumed to be passed

by value unless they are preceded by the token "ref", which signifies a pass by reference. For example:

```
proc (arg1, arg2;REF arg3);
```

8.3 Goto's

No Goto's will be allowed in the KSOS HDL designs.

8.4 Variable Declaration

All local variables must be declared and explained.

8.5 Naming conventions

All related procedures should begin with the same three letters.

All variables that begin with "TEST" are reserved for testing and should not be used.

8.6 Level of Detail

Since it is both difficult to define and follow guidelines that define the lower bound of specificity in an HDL module level design, the following is an attempt to define only the upper bound of abstraction.

The control flow of procedures should be defined. All control flow statements that will be present in the coded procedure should also be present in the design.

The interfaces in the design should be clearly defined. Interfaces consist of connections between the global environment and present environment and the actions performed with these globals objects across these connections. In order to clarify the connectional portion of interfaces, all procedure and global variables that are referenced within a procedure should be imported.

Compound statements with no references to procedures or globals may be written in prose.

KSOS Standard Headers

Ford Aerospace

Skeletal headers were produced for HDL at the CPC (module) level and the function level. These headers make use of a feature in a locally-augmented text editor that causes an interactive prompt for input ate pre-defined points. Such points are indicated in the following two pages by the "~" character.

ADDRESS: Ford Aerospace and Communications Corp.
Western Development Laboratories
3939 Fabian Way
Palo Alto, California 94303
Attention: Software Technology Dept.
Mail Stop: V02

CPCI: ~
CPC: ~
TAG: ~
MODULE NAME: ~
REFERENCES-
A Specs: ~
B5 Specs: ~
HISTORY:
AUTHOR: ~
VERSION: 1.2 of 4/8/81
MODULE TYPE:
PRIVILEGES: ~
SPECIAL NOTES: ~

```
PROCEDURE
FUNCTION
()
{
        /* ------------------------------------------------------------

        PURPOSE         Preferably a one-liner;

        DESCRIPTION {
        }

        ASSERT {
        }

        EFFECT {
        }

        PARAMETERS {
        }

        IMPORT {
        }

        CONSTANT {
        }

        TYPE {
        }

        VARIABLE {
        }

        EXCEPTIONS {
        }

        ------------------------------------------------------------ */

}
```

Modula Language Programming Standards for KSOS

John Nagle

Ford Aerospace and Communications Corporation
Palo Alto, CA 94303

1. Program Format

This section deals with the lexical layout of Modula programs and the form and content of comments.

1.1 Comments

Since there are two types of comment delimiters available (the { } pair and the (* *) pair), and since comments can be nested, it is suggested that only one set of comment delimiters be used. The (* *) pair is desirable due to its' similarity to C and HDL comments. Similarly, the { } pair is undesirable due to its' similarity with the C and HDL scope delimiters.

Specific comment rules apply to procedure declarations and USE lists, and are given in sections below.

1.1.1 Comments in record declarations

Record declarations should be written in the form

```
(*
        Comment discussing the record
*)
    type = RECORD
           field: type;          (* comment describing field *)
           field: type;          (* comment describing field *)
                .
                .
                .
           END;
```

Note that procedure and record declarations are prime targets for mechanized extraction and compilation into working documentation, so the comments in these areas should stand alone as much as possible. Imagine working from a booklet with the procedure declarations and the record declarations; these should be sufficient to use routines in other sections of the system than the one being worked on.

1.2 Indentation

Code should be indented in a manner corresponding to the static statement nesting structure, where the body of a statement construct is indented one tab stop beyond the opening and closing symbols of the construct.

1.3 White Space

Blank spaces should be used within statements and blank lines between groups of statements and comments whenever this may contribute to the readability of the code. For example, the assignment operator should be surrounded by blanks, and blank lines should be used to separate declarations and statements.

2. Program Structure

This section deals with the logical arrangement of the components of a Modula program: declarations, modules, and procedures.

2.1 Module Organization

Large programs should be partitioned so that a set of data structures and the operators which operate on that data are collected together in units called 'modules'. A subset of the data of the module and the functions which operate on the data of the module are visible (i.e. called from) outside of the module. These are said to be 'exported' from the module. Other modules which use these objects are said to 'import' them. The Modula language enforces this form of structure through its DEFINE and USE lists.

2.2 DEFINE

The 'DEFINE' statement is MODULA's version of 'exporting' procedures, variables, types, and constants. Similar organization and commenting should be used within the DEFINE statement as in the USE statement.

2.3 USE lists

The MODULA 'import' capability is called a 'USE statement'. The USE statement contains a list of variables, data types, elements of enumerated types, and procedures. Clearly, if the USE list is unordered, or unorganized, it will be impossible (ok, maybe just hard) to understand. The following USE list organization is suggested:

```
USE
        (* TYPE *)
        t1,t2,t3, ...

        (* CONST constants *)
        cc1,cc2,cc3, ...

        (* CONST variables *)
        vv1,vv2,vv3, ...

        (* VAR variables *)
        vr1,vr2,vr3, ...

        (* PROCEDURE *)
        pp1,pp2,pp3, ... ,ppX;
```

where:

VAR type variables are those which will be assigned to;

CONST type variables are those which will be read only;

Within each of the sections, a short comment on each of the 'imported' items can be very useful. It is useful to say where the object is being imported from, especially if the object is from far away and does not have a name which describes its location.

When importing an enumerated type, the type and its constants should be imported on one line in the 'types' part of the import list.

2.4 Format of Procedure Declarations

Procedure declarations shall contain a comment describing each argument. These comments shall be placed as shown below, so that extraction of all procedure definitions by a program is possible.

```
(*      Commentary describing Procedure                 *)

PROCEDURE demo(x1: integer;             (* comment describing x1 *)
               x2: char)                (* comment describing x2 *)
               : resultType;            (* comment describing result *)
```

or

```
(*      Commentary describing Procedure                 *)

PROCEDURE demo(
       x1: integer);                    (* comment describing x1 *)
```

The idea is that if the text from the last comment before PROCEDURE to the first comment after the `;' are collected, the material obtained is a good summary of how to use the procedure.

2.5 Scope of Variables and Types

It is worth reviewing the properties of types and variables after being exported. When a variable is exported via DEFINE, the variable is read only outside its home module. When a type is exported via DEFINE, the only valid operations on the type outside its home module are replacement and submission as an actual argument to a procedure.

This last is very useful. Suppose we have a module which manages open descriptors, which are defined as

```
TYPE openDescriptor: integer;
```

Within the home module, an open descriptor can be used in any integer operation. Outside its home module, the type is protected and cannot be confused with some other type whose base type is integer.

User-defined types whose base types are simple should be protected in this way, to insure that the benefits of strong typing are realized.

2.6 Preprocessing of Modula source with a preprocessor

The facilities of the C preprocessor (`cpp') are available when writing Modula code which is part of the KSOS system. Because Modula provides language facilities for functions lacking in C proper, the need for the macro processor is much less than with C code.

1. The primary use of the macro processor is to be the support of system generation. Other use of the preprocessor should be avoided. In particular, nothing should be done with the macro processor which can be done within Modula. For example, constants should be defined with Modula CONST, rather than with the `#define' of CPP. Bear in mind that it may be necessary to defend any use of the macro processor with regard to security issues.

2. The use of macros which generate code is discouraged, but not forbidden. Any such macro should bear the form of a procedure call. The syntax of Modula should not be extended with the macro processor.

3. Because `cpp' has the scope rules of C, not the scope rules of Modula, and because each program is compiled as a single unit, the names defined with `#define' must be unique across the entire program.

4. All the `#define' arguments of a program should be collected together into `#include' files.

5. A unit included via `#include' should be either a Modula module or a set of definitions with initialization. The name of the file and the name of the module should be the same, except for the `.mod' trail on the file name.

It is possible that system generation may require that system generation parameters pass through programs other than 'cpp'. The system code shall pass only through 'cpp' and the Modula compiler.

2.7 Executable Statements--Control Structures

The implementor should use recognized, well-behaved control structures. In Modula there is no alternative.

Modula does offer some useful control structures which can be used as needed. The general LOOP construct allows multiple exits from loops in a well-defined way. A CASE statement is provided and should be used when possible. The ELSE construct in the CASE statement (a York extension to the language) allows handling of a default case. Note that an undefined case in a CASE statement without an ELSE clause is a run-time detected error.

When multiple alternatives must be handled but the condition involved is too complex to handle with a CASE statement, the IF - THEN - ELSIF - THEN - END construct can be used. This avoids the excessive nesting depth induced by nesting IF - THEN - ELSE - END clauses.

3. Type Definitions and Consistency

3.1 Specific Types

3.2 Use of the type Bits

The type 'bits' should not be used for collections of flags. Arrays of boolean values or of enumeration types should be used. It is recognized that there may pressing space constraints requiring exceptions to this rule, but don't overdo it. Do not express bit constants as octal numbers; use the Modula bit string notation.

3.3 Use of Cardinal Type

Objects which are unsigned integers are preferred for verification, because it is inconvenient to lack a well-defined starting value for induction. The 'cardinal' type should thus be used when negative values are not meaningful.

3.4 Use of Enumeration Types

Integers should be used only for things which really are numbers. Things which have various nameable values should be enumeration types.

3.5 Long Integers

The types 'integer32' and 'cardinal32' may be imported when numbers longer than 16 bits are required. Only 'cardinal32' is defined within the KSOS Kernel.

Arithmetic routines are to be provided for cardinal types, as follows:

```
PROCEDURE addc32(i,j: cardinal32; VAR r: cardinal32):boolean;
PROCEDURE subc32(i,j: cardinal32; VAR r: cardinal32):boolean;
PROCEDURE mulc32(i: cardinal32; j: cardinal; VAR r: cardinal32):boolean;
PROCEDURE divc32(i: cardinal32; j: cardinal; VAR quotient: cardinal32; remainder: card
```

Note that multiplication and division take one 32-bit and one 16-bit quantity and return a 32 bit quantity. This is because most if not all multiplications and divisions is by small constants. A false return from any arithmetic operation above indicates overflow or underflow.

Signed, 32 bit arithmetic is not provided.

3.6 Representation of Sets

There is no built-in SET mechanism in Modula. For sets which are simply collections of flags, there is a rather nice representation possible.

```
TYPE                                        (* enumeration type *)
   openModes = (OMREAD, OMWRITE, OMEXCLUSIVE);
                                            (* set of open modes *)
   setOfOpenModes = ARRAY OMREAD : OMEXCLUSIVE OF boolean;

VAR
   om: setOfOpenModes;                      (* open modes *)

BEGIN

   IF om[OMREAD] THEN ....                  (* is reading allowed? *)
```

This form offers clarity and full type checking, and is preferred to the use of type 'bits'.

C Language Programming Standards for KSOS

Rance DeLong

Ford Aerospace and Communications Corporation
Palo Alto, CA 94303

The 'KSOS Implementation Plan' contains a section entitled 'Programming Standards,' which 'discusses programming standards for use with C-language programs'. This document provides additional guidelines to programmers responsible for those portions of KSOS which are written in C.

1. Program Format

This section deals with the lexical layout of the the C program code and the form and content of comments.

1.1 Commenting

Comments should not be neglected, neither should they be over-used. A format for major comments is suggested in the KSOS Implementation Plan. Right braces, when closing procedure declarations or the extended body of a statement construct, should be followed by a brief descriptive comment for orientation of the reader. Continue and break statements in extended loops should be accompanied by a comment indicating which loop is affected. Procedure declarations should have a comment header similar in form and content to the headers used for HDL procedures. At least, the 'PURPOSE' should be there.

1.2 Indentation

Code should be indented in a manner corresponding to the static statement nesting structure, where the body of a statement construct is indented one tab stop beyond the opening and closing symbols of the construct.

1.3 White Space

Blank spaces should be used within statements and blank lines between groups of statements and comments whenever this may contribute to the readability of the code. For example, the assignment operators of C should be surrounded by blanks, and blank lines should be used to separate declarations and statements.

2. Program Structure

This section deals with the logical arrangement of the components of a C program: defines, type definitions, variable declarations, function declarations, etc.

2.1 Module Organization

Large programs should be partitioned so that a set of data structures and the operators which operate on that data are collected together in units which will be called 'modules'. A subset of the data of the module and the functions which operate on the data of the module are visible (i.e. called from) outside of the module. These are said to be 'exported' from the module. Other modules which use these objects are said to 'import' them. The method which should be used, in C programs, to delimit the scope of a module, is to place the declarations and functions for the module on a single file (or several 'include' files referenced by a single file), the name of which is the module name.

2.2 Order of Definitions and Declarations

Within a module, declarations and definitions should be in the following order:

a. defines for language extending symbols (e.g. TRUE, FALSE, NIL)

b. defines for module or system parameters (e.g. BLKSIZ = 512)

c. type declarations (see section on types for details)

d. variable declarations

e. function declarations

Where all objects should be declared before referenced in the program text.

2.3 Executable Statements--Control Structures

The implementor should use recognized, well-behaved control structures. Well structured, single-entry/single-exit code is preferable in most cases to undisciplined use of the return, break, continue, and goto statements. It is recognized that, occasionally, use of these statements yields a clearer configuration than strict avoidance of them. However, implementors are expected to at least have considered the (sometimes elusive) 'totally structured' representation of a program segment before choosing an alternative representation.

3. Type Definitions and Consistency

The use of anonymous types is discouraged. Abstract types should be used, and machine representation dependencies avoided as far as possible. Strong typing will be enforced by inspection and possibly by mechanical checking. Suggestions for the declaration and use of various types follow.

3.1 Use of typedef

Types should be named by use of the 'typedef' declaration of C before they are used to declare variables of that type. Then all variable declarations (and formal parameter declarations) can be of the form: type-name declarator-list; where the declarators should be simple identifiers when the appropriate types have been previously defined.

3.2 Structure and Array Types

Structure types should be named by typedefs, not by using the structure tag in a structure or union type specifier. (Note, for self-referential structures, the tag can appear in the structure type specifier in the typedef.) Similarly, array types can be defined and named before variables are declared by using typedef and, when necessary, abstract declarators.

3.3 Enumerated and Subrange Types

For the sake of clarity, and to maintain strict typing, enumerated and subrange types should be named, though the representation of both is normally type integer. Typedef should be used to define the type name and a comment should accompany the definition which states the intended subrange or enumeration. Also, in the case of an enumeration, defines should immediately follow which equate the type member identifiers to the positive integers.

3.4 Pointers

Pointers should be used by implementors with due respect for their capacity for temptation and introducing horrendous subtle errors. Pointers should be bound to a specific type by their declaration and that they only be used to reference objects of that type. It is recommended that pointers be used in a fashion analogous to array indices, and that the results of nontrivial address calculations not be assigned to pointers.

3.5 Type Compatibility

The burden of insuring type compatibility is primarily upon the implementor, though a type checker may be available at some time. The basic criterion for type compatibility should be type name equivalence. This strictness may be relaxed for subrange types, and an identity macro, 'ORD', used for enumerated types when conversion to integer type is necessary.

3.6 Type Conversions

In addition to the ORD function, C provides some explicit type conversions. Any other necessary conversions must be supplied by the implementor in the form of macros or functions.

4. Procedure, Variable, and Constant Identifiers--Identifier Dictionary

Named objects should, where possible, have a consistent representation in the various documents, including specification, design, and implementation documents. In order to avoid name clashes resulting from C's requirement that names be unique in the first 7 characters, the KSOS UNIX Emulator will employ an Identifier Dictionary which is an M4 source file providing a long to short name translation during a precompilation pass. The Identifier Dictionary can be used to prevent and detect instances of equivocation.

Ford Aerospace

subject:
Definition of Boolean Type in KSOS UNIX*tm Emulator Code

date: January 13, 1981

from: Rance J. DeLong

TM:

ABSTRACT

While the C programming language does not have Boolean as an intrinsic type, the language does allow new types to be defined, and does support a form of logical expression. This note is intended to provide guidelines for the definition and use of a Boolean type within the Emulator code and thus represents an addendum to the C Programming Standards document.

The C language does not provide a type Boolean but in its primitive form supports integer variables, constants, and expressions which when used in contexts implying conditional tests (i.e. "logical" expressions) are interpreted to have truth values "true" or "false" depending on whether the expression evaluates to a nonzero or zero value respectively. Relational and equality operators (<, >, >=, <=, ==, !=) in the language in fact yield one and zero for "true" and "false", as do the logical operators (&&, ||, !) which take nonzero and zero values as "true" and "false" operands, this being consistent with the aforementioned interpretation of "logical" expressions.

Since the language provides no Boolean (or "logical") type, it naturally provides no intrinsic Boolean (or "logical") constants. Any choice of integer valued constants which is consistent with the language's interpretation of "logical" expressions would be suitable definitions for the symbols TRUE and FALSE. Indeed, the most natural such selection would be one and zero since it is consistent both with "logical" expression interpretation and the result values of the logical operators. However, any nonzero integer value as a definition for the constant TRUE would maintain consistency with the interpretation of "logical" expressions. The use of nonzero values other than one for TRUE would prohibit only the explicit test of a logical expression for the value TRUE (e.g. (a>b == TRUE) would be prohibited), but this is in all cases redundant and perverse anyway.

To provide a complete facility for the type Boolean a C program might include such definitions as the following:

```
typedef int Boolean;    or    typedef char Boolean;

/* Boolean Constants */
#define TRUE  1
```

```
#define FALSE 0

/* Boolean Operators */
#define AND   &&
#define OR    ||
#define NOT   !
```

Then variables may be declared as having type Boolean (e.g. Boolean end_of_file, done, error_occurred;) and such variables as well as the Boolean constants TRUE and FALSE may stand alone as Boolean expressions or in combination with the Boolean operators to form other Boolean expressions. To complete this veneer type abstraction, one should always use proper Boolean expressions wherever an expression occurs which will receive the "logical" expression interpretation (i.e. in all if, while, and for statements, conditional expressions, and assignments to Boolean variables). Thus one should not write a statement of the form:

```
while( --i ){ ... }
```

but rather,

```
while( --i != 0 ){ ... }
```

In the KSOS Unix Emulator, the Boolean constant TRUE is defined as -1 for compatibility with the KSOS Kernel. Thus it is particularly important that these guidelines be observed in Emulator C code. The only safe place to use the Boolean constants TRUE and FALSE (without appealing to knowledge of the representation — which is to be avoided) is in Boolean expressions not involving explicit tests for equality (e.g. in the assignment: end_of_file = TRUE;).

DOCUMENTATION

FACC / Palo Alto

subject: Instructions for writing KSOS Kernel Manual Pages

date: August 16, 1979

from: John Nagle
WDL Software Technology

ENGINEER'S NOTES

To achieve uniformity of appearance in manual pages, please follow the instructions given here when writing KSOS manual pages. Because these pages will be processed through the phototypesetter, the use of the specific formatter commands given is of unusual importance.

General Instructions

The manual pages for the kernel live in

/kr/sunix/kernel/doc/man

and the KSOS manual macros live in

/kr/sunix/kernel/doc/man/s.manmacros.m

These are nroff macros, not mm macros, so only nroff commands and those given in section I.9 of "Documents for use with PWB/UNIX" apply.

Some additional macros have been added to make the monospace font (which looks like 'computer output') available. This font is to be used whenever something which would appear in a user program is mentioned. All the characters in the monospace font are the same width. The new macros are

.bm "<text>" - puts out text in monospace font

works like .bd, but turns on monospace, not bold

.mn <text> - put out text in monospace, then tab

used in table entries - similar to .bn

.ag <text> - put out text in quotes and monospace

used to refer to variable names, exceptions, etc.

In general, KSOS manual pages look like UNIX manual pages. Several KSOS manual pages have been created and may be used as guides.

Remember that tables must line up on the phototypesetter. Use .tl as required. If all else fails, go to no fill mode and use the monospace font - there everything will line up as it does on the printer.

NAME section

```
.sh NAME
<shortname> [<longname>] - <1 line summary>
```

SYNOPSIS section

This is the hard one. Modula, C, and Assembler calling sequences must be given. For uniformity of appearance, the following rules should be observed.

1. Everything which represents programming language source statements is to be in the monospace font. Each line should begin with ".bm" and the text on the line is in double quotes.

2. Each section (Modula, C, and Assembler) is separated by a blank line. There are no blank lines within sections. If a separator is required within a section, "bd ..." should be used.

3. The calls are given in the order Modula, C, Assembler

4. In the Modula section, a variable is declared of the required type for each argument and then used in the call. Variables which are input-only are declared as CONST, and result variables are declared as VAR. See the example. Type definitions do NOT appear here, but will be provided in an introduction to the kernel calls.

5. In the C section, each variable is declared, and the function is called. Arguments which are bigger than 2 words or results are passed as pointers; others are simply passed as values.

6. In the assembler section, code to put the arguments on the stack is provided, and the sequence ends with an emt to the appropriate routine. A numeric value for the entry is NOT to be provided at this time. In general, entries and exceptions should not be given numeric values at this time on the manual page.

DESCRIPTION section

Up to the author. Names of variables or exceptions should be given as

```
.ag "<variable>"
```

which puts them in quotes and in the monospace font. Names of other calls or KSOS commands should be given in the form

.it "<command> (II)

This will look like the UNIX manual.

SEE ALSO section

In this section, names of other commands are NOT given in italics, since nothing else appears here.

EXCEPTIONS section

A table of possible exceptions should appear as shown in the example. Do not give numeric values at this time; these may be included at a much later date, and if so will be included mechanically. The X code for the exception should begin in column 1 of the NROFF source to aid this effort.

Final words

I regret having to specify all this minor detail, but to achieve uniform appearance it is a must; and uniform appearance is much to be desired here. Incidentally, bear in mind that this will be the most-closely-read and most-used documentation in all of KSOS, and may still be in active use in the 1990s.

John Nagle

Appendix - An example

TESTING

Unit Test Plan for the Department of Defense Kernelized Secure Operating System (KSOS)

Ford Aerospace & Communications Corporation

Western Development Laboratories Division

Palo Alto, California 94303

1. PURPOSE

The purpose of this test plan is to provide the detailed information required for unit testing of KSOS software.

Unit testing shows how well a unit, standing alone, matches its design. Unit testing is just one test which the software must pass through. One common error is to think that everything must be tested at this stage. Unit testing has a limited objective of testing the modules that make up a unit in a stand alone environment. Other forms of testing test against other objectives. Integration testing tests the interfaces between units. Category I testing tests each CPCI for functionality, and Category II testing tests the functionality of the entire system.

The cost of correcting an error increases dramatically as its discovery is delayed through the testing cycle. In order to meet the tight KSOS implementation schedule, as many errors as possible must be found during unit testing.

This unit test plan applies to all modules of the 3 KSOS CPCI's: Security Kernel, UNIX Emulator, and Non-Kernel Security Related Software.

2. COMPUTER PROGRAM TEST OBJECTIVES

The minimum acceptable unit test completion criterion is the exercising of every logical branch in the module. The effectiveness of this is discussed in detail in WDL's Software Engineering Practices Manual. Additional objectives of unit testing are to demonstrate internal unit integrity, data handling over the specified range of input values, error recognition and handling, and interface integrity between modules within the units. The unit test procedure shall insure that

1. all logical branches of each module are exercised at least once,
2. WHILE and REPEAT loops which are controlled by a loop counter are exercised zero and "many" times, WHILE and REPEAT loops which are not controlled by a loop counter are exercised zero and at least one time,
3. each module is limit tested, and
4. the functional requirements allocated to the unit are demonstrated.

3. IMPLEMENTATION

3.1 REQUIRED DOCUMENTATION

The following documents shall be added to the Unit Develoment Folder as they are completed. Since the KSOS documentation is kept on line, the Unit Development Folder exists in the form of a directory called /kr/sunix/<cpci>/<cpc>/<cp>/udf. One item in the udf directory should be a "test" directory which contains the Unit Test Procedure, any programs, shells, or data files necessary to run the test, and test results.

3.1.1 Unit Test Procedure

Test procedures, which are written by the programmers, are the procedures which are to be followed in testing a module. They specify the test data to be used, the expected results, and any intermediate results which must be printed in order to verify the results of the test. A suggested format for recording the test procedures is shown in Figure 1.

TEST PATH NUMBER	TEST INPUT	EXPECTED OUTPUT	TEST RESULTS
1	A=0 B=0 C=Xstart TRNL=727	After first pass through load loop, FREE=727. At end of load loop, FREE=0.	
2	B=8 INV=32 alpha=33, LEN=0 from REGEN 6 from GETMAX seq.error from TRANS	B=INV	
...	...	...	

Figure 1. Example test procedures.

Column one of this format specifies the test path number. Column two specifies the input data, the results returned by the stubs, and the source of any input data required to run the test.

The expected output is indicated in column three. The output is normally indicated by specifying the value of selected variables at the end of the test run. However, the expected outputs can be designated by showing the relationship between variables. For example, an expected result might be that A=B. If testing requires the printing of intermediate results, these printed results are indicated in this column.

Column four contains the actual test results. If the expected output is a relation such as A=B, the actual result should be shown. Otherwise a pass/fail indication is sufficient.

Other important information to be indicated along with the test procedures is the test environment. The test environment includes such things as the module version and the data base version. This information facilitates the possible re-running of the tests.

3.1.2 Test Results

The Test Results shall be in the form of annotated listings of the actual results of the tests as outlined in the procedure. They shall include the date of the test, the name of the person who ran it, and any relevant comments.

3.2 TOP DOWN V.S. BOTTOM UP

3.2.1 Strategy

One way the testing of modules within a unit can be approached is top down. The process begins with the top level module. Modules below this level are replaced by "stubs". Each stub performs two functions. First, a stub must return a result and perhaps create side effects which will allow meaningful execution of the calling module to continue. It is usually sufficient for the stub to return constants for output although it may be necessary at times for the stub to solicit information from the operator or to obtain its results from a script file. Secondly, a stub may print pertinent information to aid the evaluation of the software's performance. For example, the stub may print a message stating who called the stub and what the inputs and outputs were.

After testing this portion of the software and determining that the major interfaces are working correctly, another level of logic is added. This procedure continues until the entire system is tested and integrated, one level at a time.

Since KSOS is an operating system, it depends rather heavily on some of the lower level modules (I/O drivers, for example), and therefore not all units can be tested purely top down. These critical modules can be integrated into the unit -- or tested alone at first -- on an as-needed basis. Also, the testing of time critical parts of real time software may proceed somewhat differently from the top down testing outlined previously. It is sometimes necessary or convenient to test execution paths or threads that exercise the time critical parts of the design. In this case, the modules necessary to exercise the thread are written and tested together. In order to accomplish this thread testing, a test harness may have to be written to enable the exercise of the interrupts and to capture the timing and performance data during the thread test. Whenever possible, modules should be tested using the already tested higher modules as a driver.

The preferred method of unit testing is top down. This exercises the top level interfaces first. This is important since top level interface errors are the most difficult errors to fix. In addition, this procedure minimizes the amount of throw away driver code which must be written.

3.2.2 Test Drivers

One suggested driver for a module that must be tested independently of its natural driver is the following test frame controlled by a terminal or script file.

```
{
MODULE DESCRIPTOR:  Test Frame Skeleton
HISTORY:
  AUTHOR:   Richard B. Neely
  VERSION:  1.2 of 6/27/79
  MODULE TYPE:
  The character '<&>' is used in place of the module name to be
  tested, and the construction <&---&> is to be replaced with a
  more complex item.
}

MODULE <&>testFrame;

#include "/usr/mod/seqmodio.mod"        {UNIX I/O interface
                                                -- must be first}
#include "<&>.mod"                      {Module to be tested}

CONST
    inFile  = '<&>_1.fs';  {Script for test — up to <&>_99.fs}
    outFile = '<&>_1.fl';  {Log for test — up to <&>_99.fl}

VAR command: char;   {Commands of one character are assumed}


BEGIN

reset(inFile);          {I/O initialization}
rewrite(outFile);

writes('Test frame v. 1.2 for <&> v. '); writes(&version); newline;
  {Prints both the version of the test frame and of the module}

<&initialization&>      {i.e., of the module — if applicable}

LOOP
    readc(command);
    WHEN eof DO      {In case no quit command}
         newline; writes('END of input'); newline
    EXIT
    WHEN command = '<&quit cmd&>' DO
                           {This allows some standard termin.}
         <&termination&>   {again, of the module}
    EXIT
    CASE command OF
         '<&cmd1&>':       {One of these for each command}
             BEGIN
```

```
            <&fcn1&>
            END;
        <&etc.&>
    END
END

END <&>testFrame.
```

In addition to commands to test the module and print results, the test frame could accept a command to write a comment from the terminal or script file. These comments are used to automatically annotate the test output without requiring a separate manual edit.

Another possible driver would be a less general program which initializes data and calls the module as many times as needed, with no intervention or script file. It may also be desirable for this program to put comments into the output file.

No single type of driver is appropriate for all types of units/tests. These are suggested starting points for writing a driver tailored to the needs of the unit under test.

3.2.3 I/O Package for Modula

The I/O procedures are in the file "/kr/sunix/include/strio.h"

setStream - All I/O goes to the files set by the setStreams call. "setStreams" sets the input and output streams. If not set, the standard input and output are assumed.

getChar - The procedure getChar will read one character from the in-stream.

```
getChar (VAR c : char) : boolean;
```

A value of false is returned upon eof.

getString - The procedure getString will read the in-stream until the break character is encountered or until the passed array is full.

```
getString (breakchar : char;
           VAR string : ARRAY integer OF char) : boolean;
```

A value of false is returned if a eof is encountered. Note that the break character is lost from the input stream.

getLine - The procedure getLine will read the in-stream until a newline (octal 12) is encountered and place the characters read in the passed array. However the new-

line is replaced with a null (octal 0).

getLine (VAR string : ARRAY integer OF char) : boolean;

A value of false is returned upon eof.

getDec - The procedure getDec will read a decimal number from the in-stream. Leading signs are accepted.

getDec (VAR number : integer; VAR break : char) : boolean;

A value of false is returned upon eof or if attempting to read non-numerical input. The parameter break is filled with the first non-numeric character encountered.

getOct - The procedure getOct will read a octal number from the in-stream. Leading signs are accepted.

getOct (VAR number : integer; VAR break : char) : boolean;

A value of false is returned upon eof or if attempting to read non-numerical input. The parameter break is filled with the first non-numeric character encountered.

putChar - The procedure putChar will output the passed character to the out-stream.

```
putChar (character : char);
```

putNl - The procedure putNl will ouput a newline to the out-stream.

```
putNl;
```

putString - The procedure putString will output the passed string to the out-stream. The length of the string is taken to be the the length of the array or the characters in the array up to and not including the first zero (null).

```
putString (string : ARRAY integer OF char);
```

putDec - The procedure putDec will output the passed number in decimal representation.

```
putDec (number : integer);
```

putOct - The procedure putOct will output the passed number in octal representation.

```
putOct (number : integer);
```

3.3 HOW TO EXERCISE EVERY BRANCH

3.3.1 Methods

There are two basic approaches to the problem of exercising every logical branch. One is analytic; the other is empirical.

3.3.1.1 Analytic Method

The analytic method has two planning steps. First test paths must be found which will cover all branches, then data must be found which will exercise the paths.

3.3.1.1.1 Test Paths

The first step of the analytic method is to list vertically the decision language constructs (such as IF, WHILE, and CASE) in the order in which they appear in the detailed design description. This list defines all branches within the program. The line number where each construct appears should also be included. The first two columns of Figure 3 illustrate a suggested format for organizing this information as extracted from the Modula in Figure 2.

Test paths are created by following hypothetical execution histories. At each decision point, one tries to choose a branch that has not been tested by any of the previous test paths. The set of test paths is complete when every branch is exercised at least once.

The test paths can be recorded in vertical columns parallel to the list of decision constructs as shown in Figure 3. If a test path exercises an IF-THEN-ELSE construct, then an X is placed either in the IF row if the true branch is taken or in the ELSE row if the false branch is taken. Similarly, X's are placed in the appropriate row for every construct exercised by the path.

Sometimes a path needed to test construct A passes through previously tested construct B. Instead of randomly putting an X by a branch of B, it may be better to indicate the "don't care" condition with a "-" by all branches of B. This allows more flexibility in choosing test data than the random choice and will shorten the time required to choose test data for complex code where "don't cares" occur.

Special note on loops: Both branches of a WHILE statement can be tested by making the predicate true initially and letting the loop change something so it becomes false. Therefore, it is possible to exercise both branches with one test path. However, KSOS is imposing an additional requirement. A loop which is controlled by a "counter" may behave correctly when subjected to the above test (i.e. the loop body is executed once), but the loop may not work for zero or many iterations. Therefore for KSOS, the skeleton should show "zero" and "many" iterations for WHILE loops whenever possible. (The "exactly one iteration" case is covered by "many", but generally cannot be substituted for it in this context.)

Since the body of a REPEAT is executed before the test, the skeleton should show "one" and "many".

```
1           BEGIN
2             s1
3             WHILE p1 DO
4               IF p2 THEN
5                 LOOP
6                   s2
7                   WHEN p3 DO
8                     s3
9                     EXIT
10                  s4
11                  WHEN p4 EXIT
12                  s5
13                END
14              ELSE
15                REPEAT
16                  s6
17                UNTIL p5
18              END;
19              CASE p6 OF
20                    1:
21                       BEGIN
22                       s7
23                       END
24                    2:
25                       BEGIN
26                       s8
27                       END
28                ELSE:
29                       s9
30              END;
31              s10
32            END
33          END;
```

FIGURE 2.

For simplicity, only the success branch of each WHEN decision is explicitly shown. Failure of a WHEN is implied by the success of a subsequent WHEN in the same LOOP. The case where all WHEN's fail and the LOOP begins executing again is the only case in addition to success of each WHEN required to show that all branches have been exercised. Therefore, LOOP should be shown as LOOP>0 to show a minimum of one complete pass through the loop. LOOP, with its choice of exit points is designed to be controlled by several independent predicates, making the "zero"/"many" criteria less conclusive.

When a loop contains items which show up in the skeleton, sometimes it's convenient to test several branches inside the loop with a single path. For example, both branches of the IF-THEN-ELSE in Figure 4 might be exercised by one invocation of the WHILE. The loop body executes twice, modifying the data checked by the IF. Although this is an efficient path, its representation on

LINE NUMBER	CONSTRUCT	TEST PATH 1	2	3	4
3	WHILE0	X			
	WHILE>1				X
4	IF		X	X	
5	LOOP>0		X		
7	WHEN		X		
11	WHEN			X	
14	ELSE				XX
15	REPEAT1				X
	REPEAT>1				X
19	CASE 1		X		-
	2			X	-
	ELSE				X-

Figure 3.

the chart can be unclear. In a simple case like this, it may not matter whether or not the ELSE is executed the first time through the loop and the IF the second time, but in a more complex case, the column of X's could be confusing. It might be possible to test the example in Figure 3 with only two paths as shown in Figure 5. Finding tricky paths like #2 can be a waste of time, but if such a path is natural and obvious to the programmer, it should not be overlooked. Generally, branches are executed in the order in which they appear on the page, top to bottom. The confusion in a case like this arises from them being executed "out of order." This confusion can be eliminated by starting a new column every time a "higher" branch is executed after a "lower" branch. Path #3 in Figure 5 eliminates the guesswork needed to interpret #2.

Ideally, the skeleton should come from the HDL before coding, but since HDL doesn't map to Modula as well as we would like, the Modula source code will generally be used. The test completion monitor described below produces the information needed to complete this skeleton. The tool compiles a list of the control constructs in the Modula source, including the line and probe numbers.

3.3.1.1.2 Test Data

In addition to defining test data to execute every test path, test data must be generated to show that the unit's functional and design requirements are met, and to test the unit's response to out of range input. This data is generated after the code is written since only then does one know the exact form of the tests which control each branch.

The easiest way to construct test data which exercise every test path involves an iterative trial and error process. An initial estimate is made of the input data required to exercise the test path under consideration. When a branch is found that goes the wrong way, the estimate is then modified to reflect the knowledge gained so that the required branch is executed. This

LINE NUMBER	CONSTRUCT	TEST PATH 1	2
3	WHILE0	X	
	WHILE>1		X
5	IF		X
8	ELSE		X

Figure 4.

process continues until the entire path is exercised.

During the generation of test data, one may find that there is no valid input data which will exercise some of the specified paths. These paths are called phantom paths. Phantom paths are usually created when independent tests in the algorithm are not logically independent. For example, in a program which classifies integers, multiples of two and powers of two which are greater than ten might be tested for independently, but only multiples of two are powers of two greater than ten in the real world.

There are two ways of dealing with phantom paths. One way is to modify the set of test paths so as to eliminate them. The second way is to modify the module itself in order to force the appropriate branch. This modification is done by inserting unconditional GOTOs or by modifying the test conditions. Whenever possible, the first method is preferred over the second.

3.3.1.2 Empirical Method

The empirical method relies heavily on the Probe. The data is not available for inclusion in the test procedure until after the test has been run once successfully because it is chosen interactively. The module is tested with a collection of data chosen because it might work. The Probe then reports how well it worked, and more data is tried until the probe reports that all branches have been exercised. Once a good data set has been chosen, it should be written into the test procedure, along with test results so the test can be repeated as the module gets revised. It is important that the expected output actually be determined for each test input.

Since the analytic approach is very powerful, it should be used whenever possible. Unfortunately, it becomes cumbersome when used on large complex modules, so common sense suggests the use of the empirical approach in those cases.

3.3.2 Probe Usage

The initial step in using the test completion monitor is to insure that the MODULA module or program to be instrumented is syntactically correct. The modula compiler can be used to perform this check. This step is necessary since the tool accepts a super set of modula and does little error checking.

LINE NUMBER	CONSTRUCT	TEST PATH 1	2	3
3	WHILE0	X		
	WHILE>1		X	X
4	IF		X	XX
5	LOOP>0		X	X
7	WHEN		X	X
11	WHEN		X	X
14	ELSE		X	XX
15	REPEAT1		X	X
	REPEAT>1		X	X
19	CASE 1		X	X -
	2		X	X -
	ELSE		X	X-

Figure 5.

The syntactically correct source can then be instrumented using tcmon which is located in /kr/sunix/bin. The command line has the form:

tcmon flag filename

The optional flag argument allows the user to specify the environment in which the instrumented source will be run. The possible flags include -F for a module exercised with a test frame on the UNIX system, -B for a program to be run on the bare machine, and -T for a module exercised with a test frame on the bare machine. The default is a program which runs on the UNIX system.

All four "versions" of the tool tcmon produce two outputs; the instrumented source and a skeleton of the source. The instrumented source is written to the file source_pr and consists of the MODULA source with software probes and supporting software inserted. While the sources produced by the different versions of tcmon are basically the same, there are a few differences to compensate for the different environments. When the -B flag is used, for example, a different probe handling module is used. On the other hand, when the -F flag is used, the user must include a call to testwrap ("testwrap;") in the test frame and place it in the necessary DEFINE lists of the frame. Testwrap insures that all of the monitor log is printed at the end of the test run. This clean up is necessary as the monitor log output is buffered. The call, therefore, should be placed such that it is executed after the source has been exercised. It is also assumed that the test frame will make use of the sequential modula I/O.

The other output, source_sk, is a skeleton of the modula source which indicates all branches which exist in that source. The skeleton is made up of a list of the probes placed in the source by tcmon. Each entry in the list consists of the number of the probe, the line number where the probe is located in the source, and the type of probe.

After tcmon is executed, the instrumented source, source_pr, is then compiled by the user with the necessary compiler flags. Test cases may then be executed. Execution of source_pr will cause a monitor log, source_ml, to be produced in addition to the outputs of the original source. The first entry of this log is the total number of probes were placed in the source. The other entries are the numbers of the probes as they were executed. If source_pr is run more than once, source_ml should be moved to preserve the log.

Source_ml can either be examined as a trace or be analyzed by the data reducer, data_reduce. Data_reduce examines one or more monitor logs and reduces the information they contain to a readable summary. When the user executes data_reduce, it asks him for the name of a monitor log. This query is repeated when processing on the monitor log is completed. When all logs have been processed the user should enter "EOF". Data_reduce then produces the summary of the logs and places it in source_sm. This summary consists of three parts: a list of all probes and the number of times they were executed (a profile of the run), a list of those probes which were not executed, and two sums indicating the number of probes executed and not executed.

An example of the use of this tool is shown below. The file t4a1.mod is compiled to insure that it is syntactically correct. Tcmon is then executed. When it is finished, the instrumented source, source_pr, is compiled and then executed producing the monitor log source_ml. data_reduce is then executed with source_ml as its input.

```
% mod t4a1.mod          ;:just to confirm syntax
% tcmon t4a1.mod        ;:produces source_pr and source_sk
    1214
    #
    {4}                     DEFINE testwrap;
    {5}                     USE write, creat;
    {29}                    {total testprobes}
    {testwrap call placed 2 lines up}
    {49}    END t4a1.
    1177
% mod source_pr         ;:compile probed source
% a.out                 ;: execute probed object file,
                         : produces source_ml
% data_reduce
    please enter name of the monitor log
    if done just type EOF
source_ml
    please enter name of the monitor log
    if done just type EOF
EOF
%
```

Tcmon contains some d editor scripts which produce text such as that in the above example. This text is different for each version as the changes made by the scripts are the adjustments required for the particular environment. A sign of trouble, however, is a question mark in this text.

The example above produces four files; source_pr, source_sk, source_ml, and source_sm. source_sk and source_sm are shown below.

Probe Number	Line Number	Probe Type
1	13	BEGIN
2	16	END OF BODY
3	22	BEGIN
4	25	END OF BODY
5	31	BEGIN
6	48	END OF BODY

source_sk

probes in the source	number of times executed
probe 1	3
probe 2	3
probe 3	3
probe 4	3
probe 5	1
probe 6	1

probes not executed

total number of probes executed = 6
total number of probes not executed = 0

source_sm

3.4 SOURCES OF TEST INPUT DATA

KSOS has no files of "standard" test data for unit testing. All test data will be chosen by the programmers.

4. REFERENCES

- "KSOS Program Product Specifications (Type C-5)" Ford Aerospace and Communications Corporation, Palo Alto, California (February 1979)

- "KSOS Configuration Management Plan" Ford Aerospace and Communications Corporation, Palo Alto, California (August 1978)
- "KSOS Implementation Plan" WDL-TR7799, Ford Aerospace and Communications Corporation, Palo Alto, California, (March 1978)
- "Huang, J.C. An Approach to Program Testing." ACM Computing Surveys, 7(3), 113ff.
- "WDL Software Engineering Practices Manual" Ford Aerospace and Communications Corporation, Palo Alto, California (February 1979)


FILMED
3-8